Piotr Gołoś Paweł Szostek

Wnioskowanie w rozproszonej bazie wiedzy Dokumentacja projektowa

1. Problem

Celem projektu jest zaprojektowanie i zrealizowanie aplikacja do wnioskowania w grupie agentów. Algorytm wnioskowania ma uzyskać dokładnie te same rezultaty, co w przypadku wiedzy scentralizowanej będącej sumą teorii poszczególnych agentów. Proces wnioskowania będzie rozproszony i żaden agent nie ma kontroli nad jego całością.

2. Algorytm

2.1. Założenia

— Algorytm opiera się na komunikacja między agentami. Agenty nie akumulują wiedzy od innych agentów, a proces wnioskowania opiera się na przekazywaniu wiadomości w systemie powiązanych agentów.
— Implementowany algorytm zakłada, że przechowywana baza wiedzy jest spójna. W przypadku zaistnienia sprzeczności w bazie wiedzy nie jest zapewniona poprawność wnioskowania.
— Algorytm zakłada, że w trakcie pracy systemu poszczególne agenty mogą się odłączać oraz nowe agenty mogą się łączyć z systemem. Agenty dynamicznie uaktualniają swoją bazę połączeń (każdy agent przechowuje informacje o agentach, z którymi może prowadzić komunikację),
— Komunikacja między agentami będzie oparta na zadanym grafie połączeń. Dowolna para agentów będzie się w stanie komunikować między sobą wtedy, kiedy będzie istniało między nimi połączenie w grafie sąsiedztwa.
— Agenty odbierają klauzule od innych agentów i uwzględniają je w swoim procesie wnioskowania.
— Do określania lokalnych konsekwencji literałów otrzymanych z zewnątrz używana będzie rezolucja, np:

$$\frac{\neg a \vee b, a \vee c}{b \vee c} \tag{1}$$

2.2. Komunikacja międzyagentowa

W projektowanym programie implementowany będzie mechanizm komunikacji, którego celem jest wymiana wiadomości pomiędzy agentami, prowa-

dząca do rozstrzygnięcia, czy zapytanie zadane z zewnątrz systemu jest prawdziwe czy fałszywe. Algorytm zakłada trzy typy wiadomości, z których każda wyzwala jedną z trzech procedur. Wiadomości mają postać m(*Sender*,*Receiver*, *message_type*, *hist*, *l*), gdzie:

Sender to agent wysyłający wiadomość,

Receiver to agent odbierający wiadomość,

message_type to stała ze zbioru {FORTH, BACK, FINAL}, określająca typ wiadomości,

hist to historia zapytań,

l to literał, będący dowodzony w bierzącej gałęzi wnioskowania. W każdej gałęzi wnioskowania przesyłany jest tylko jeden literał.

Do przesyłania obiektów wewnątrz wiadomości zostanie wykorzystany mechanizm oferowany przez Jade, wykorzystujący ontologie (konieczność zdefiniowania klasy wiadomości i klasy ontologii, mapującej obiekty do slotów wiadmości).

W poniżej zamieszczonym pseudokodzie wspomnianych procedur przyjęto następujące oznaczenia:

BOTTOM A CHINY NIE WIEM

Resolvent(p, Agent) to zbiór konsekwencji literału p w bazie wiedzy agenta *Agent*

LOCAL to zmienna pomocnicza przechowująca otrzymaną klauzulę i jej lokalne konsekwencje

ACQ(l, Agent) to zbiór Agentów, które mają połączenie z agentem *Agent* i które współdzielą z nim klauzulę *l*

Algorithm 1 Procedura wywoływana przy odbiorze wiadomości typu Forth

ReceiveForthMessage(m(Sender, Self, forth, hist, p))
if (¬ p,_,_) ∈ hist **then**
  send m(Self, Sender, back, [(p, Self, □)|hist], □)
  send m(Self, Sender, final, [(p, Self, **true**)|hist], **true**)
else if p ∈ Self or (p, Self,_) ∈ hist **then**
  send m(Self, Sender, final, [(p, Self, **true**)|hist], **true**)
else
  LOCAL(Self) ← {p} ∪ Resolvent(p, Self)
  if □ ∈ LOCAL(Self) **then**
    send m(Self, Sender, back, [(p, Self, □)|hist], □)
    send m(Self, Sender, final, [(p, Self, **true**)|hist], **true**)
  else
    LOCAL(Self) ← {c ∈ LOCAL(Self) | all literals in c are shared}
    if $LOCAL(Self) = \emptyset$ **then**
      send m(Self, Sender, FINAL, [(p, Self, **true**)|hist])
    end if
    for all c ∈ LOCAL(Self) **do**
      for all l ∈ c **do**
        BOTTOM(l, [(p, Self, c)|hist]) ← **false**
        for all RP ∈ ACQ(l, Self) **do**
          FINAL(l, [(p, Self, c)|hist],RP) ← **false**
          send m(Self, RP, forth, [(p, Self, c)|hist], l)
        end for
      end for
    end for
  end if
end if

Algorithm 2 Procedura wykonywana przy odbiorze wiadomości typu Back

ReceiveBackMessage(m(Sender, Self, back, hist))
{hist ma postać [(l', Sender, c'), (p, Self, c)|hist']}
BOTTOM(l', [(p, Self, c)|hist'])← **true**
if ∀ l ∈ c, BOTTOM(l, [(p, Self, c)|hist']) = **true then**
  if hist' = ∅ **then**
    U ← User {odbiorcą wiadomości będzie użytkownik}
  else
    U ← the first peer P' of hist' {odbiorcą będzie poprzedni agent z historii}
  end if
  send m(Self, U, bacl, [(p, Self, c)|hist'])
  send m(Self, U, final, [(p, Self, c)|hist'])
end if

Algorithm 3 Procedura wykonywana przy odbiorze wiadomości typu Final

ReceiveFinalMessage(m(Sender, Self, final, hist))
{hist jest postaci [(l', Sender, **true**), (p, Self, c)|hist']}
FINAL(l', [(p, Self, c)|hist'], Sender) ← **true**
if ∀ c^* ∈ LOCAL(Self) and ∀ l ∈ c^*, FINAL(l, [(p, Self, c^*)|hist'], _) = **true then**
  if hist' = ∅ **then**
    U ← User {odbiorcą wiadomości będzie Użytkownik}
  else
    U ← the first peer P' of hist' {odbiorcą będzie poprzedni agent z historii}
  end if
end if
send m(Self,U, final, [(p, Self, **true**)|hist']) {usuń nadawcę wiadomości z historii i prześlij wiadomość do poprzedniego agenta w tej gałęzi wnioskowania}

3. Implementacja agenta

Wszystkie agenty będą homogeniczne i będą implementowały te same zachowania. Agenty będą się różniły wiedzą i połączeniami, które będą mu podawane na etapie tworzenia.

3.1. Zachowania agenta

Każdy agent będzie posiadał zachowanie cykliczne, które będzie polegało na odbieraniu wiadomosci od innych agentów oraz na reakcji na nie. Obsługa wiadomości będzie implementowana jako pojedyncze zachowania, uruchamiane przy odbiorze odpowiedniego typu wiadomości. Poniżej zamieszczono szkic implementacji zachowania:

```
class ServeMessages extends CyclicBehaviour {
    public ServeMessagese(Agent a) {
        super(a);
    }

    public void action() {
        ACLMessage msg = this.receive();
        if (msg != null) {
            // otrzymano wiadomość - sprawdź jej typ i uruchom procedurę
            String content = msg.getContent();
            type = parseType(content)
            if(type == FORTH)
                addBehaviour(new HandleForthMessage(this, msg));
            else if(type == BACK)
                addBehaviour(new HandleBackMessage(this, msg));
            else if(type == FINAL)
                addBehaviour(new HandleFinalMessage(this, msg));
```

```
            else
                throw UnknownMessageTypeException();
        }
    }
}

class HandleForthMessage extends OneShotBehaviour {
    ACLMessage request;

    public HandleForthMessage(Agent a, ACLMessage msg) {
        super(a);
        this.request = msg;
    }
    public void action() {
        //w tym miejscu implementacja algorytmu zamieszczonego powyżej
    }
}

class HandleBackMessage extends OneShotBehaviour {
    ACLMessage request;
    public HandleBackMessage(Agent a, ACLMessage msg) { /* ... */ }
    public void action() {
        //...
    }
}

class HandleFinalMessage extends OneShotBehaviour {
    ACLMessage request;
    public HandleFinalMessage(Agent a, ACLMessage msg) {/*...*/}
    public void action() {
        //...
    }
}
```

Procedura `ReceiveForthMessage` jest wyzwalana poprzez otrzymanie wiadomości m(Sender,Receiver, forth, hist, l) wysyłanej przez agenta Sender do agenta Receiver: na żądanie agenta Sender, z którym agent Receiver współdzieli zmienną l, przetwarzany jest literał l.

Procedura `ReceiveBackMessage` jest wyzwalana przy odebraniu wiadomości m(Sender,Receiver, back, hist, r) od agenta Sender do agenta Receiver i przetwarza odpowiedź r udzieloną przez agenta Sender (dla literału l).

Procedura receiveFinalMessage jest wyzwalana przy oderbaniu wiadomości m(Sender,Receiver, final, hist, true) od agenta Sender do agenta Receiver, w której komunikuje, że liczenie konsekwencji dla literału l (ostatnio dodany w historii) jest zakończone.

Według oryginalnego algorytmu zapytanie q jest zadawane przez użytkownika agentowi P poprzez wysłanie wiadomości m(User, P, forth, $\emptyset$, $\neg$q) i

jest zakończone sukcesem w przypadu odebrania wiadomości m(P, User, back, hist). Jeżeli zapytanie nie może być dowiedzione przez system, użytkownik jest ostatecznie informowany poprzez wysłanie wiadomości m(P, User, final, hist) bez wysłania odpowiedniej wiadomości zwrotnej. W projektowanej aplikacji zadawanie zapytania będzie zmodyfikowane

Algorytm będzie implementowany w agencie poprzez zachowanie cykliczne, które będzie polegało na odbieraniu wiadomosci od innych agentów oraz na reakcji na nie. Poniżej zamieszczono szkic implementacji zachowania:

```
private class ServeMessages extends CyclicBehaviour {
    public void action() {
        ACLMessage msg = myAgent.receive();
        if (msg != null) {
            // otrzymano wiadomość - sprawdź jej typ i uruchom procedurę
            String content = msg.getContent();
            type = parseType(content)
            if(type == FORTH)
                receiveForthMessage(msg);
            else if(type == BACK)
                receiveBackMessage(msg);
            else if(type == FINAL)
                receiveFinalMessage(msg);
            else
                throw UnknownMessageTypeException();
        }
    }
}
```

3.2. Struktura agenta

W każdy agencie przechowywana jest wiedza w postaci zbioru klauzul. Każda klauzula ma postać alternatywy literałów. Może też być pojedynczym literałem albo klauzulą pustą. Implementacja: wzorzec kompozytu.

W każdym agencie przechowywane są dwie struktury danych związane z procesem wnioskowania: -cons(l, hist) przechowuje konsekwencje klauzuli l obliczone w gałęzi wnioskowania odpowiadającej historii hist -final(q, hist) jest prawdziwe, kiedy propagacja zapytania q wewnątrz gałęzi wnioskowania, odpowiadającej historii hist jest zakończone

Historia to list krotek (l, P, c), gdzie:
l to literał, będący częścią klauzuli c z poprzedniej krotki z listy,
P to agent,
c to klauzula, będąca konksekwencją literału l w agencie P.
Historia jest przesyłana w każdej wiadomości i pozwala ona się zorientować, jak wygląda dana gałąź wnioskowania. Dzięki niej możliwe jest wykrycie cykli

4. Interfejs

4.1. Baza wiedzy

Baza wiedzy dla programu będzie przechowywana w postaci pliku tekstowego. Dane z niego będą używane w momencie tworzenia agentów i będą im przekazywane jako argumenty wywołania.
Plik jest podzielony na dwie części: w pierwszej części w każdej linii zawarte są informacje na temat bazy wiedzy rozproszonej pomiędzy agentami. Druga część zawiera informacjeo o połączeniach między agentami. Poniżej gramatyka w formiacie EBNF opisująca składnię pliku z danymi:

```
dane: wiedza_agentow '\n' połączenia_agentów

wiedza_agentow: wiedza_agentow wiedza_agenta | wiedza_agenta
wiedza_agenta: identyfikator ':' klauzule '\n'
klauzule: klauzule ', ' klauzula | klauzula | ""
klauzula: klauzula '+' literał | literał
identyfikator: LICZBA

połączenia_agentów: połączenia_agentów połączenia_agenta
| połączenia_agenta
połączenia agenta: identyfkator ':' lista_identyfikatorów '\n'
lista_identyfikatorów: lista_identyfikatorów ', ' identyfikator
| identyfikator | ''

literał: NEGACJA LITERA | LITERA
NEGACJA: '~'
LITERA: {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o',
'p','q','r','s','t','u','v','w','x','y','z','A','B','C','D','E','F',
'G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W',
'X', 'Y', 'Z'}

Przykładowy plik z danymi:
1:a+b+c, b, d+~a
2:c+a, ~e
3:~a
4:e+a

1:1, 2, 3
2:1
3:1, 4
4:3
```

Baza wiedzy zawarta w pliku nie może byc sprzeczna. Informacje o połączeniach między agentami muszą być spójne. Połączenia są dwukierunkowe. Jeśli występuje połączenie od agenta A do agenta B, to musi też istnieć połączenie od agenta B do agenta A.

4.2. Instancjacja agentów

Agenty są instancjowane z wykorzystaniem klasy jade.Boot. Wiedza agenta jest przekazywana w formie stringa jako argument konstruktora klasy.

4.3. Obsługa zapytania

Każde zapytanie zadane agentowi jest przez niego rozkładane na drzewo rozbioru gramatycznego, w którym w liściach przechowywane są pojedyncze literały, natomiast w węzłach nie będacych liśćmi przechowywane są operatory logiczne, łączące zapytania złożone przechowywane w poddrzewach podłączonych do danego wierzchołka. Zapytania złożone będące alternatywą logiczną są prawdziwe, jeśli co najmniej jedno z podzapytań jest prawdziwe. Koniunkcje logiczne są prawdziwe, jeśli wszystkie podzapytania są prawdziwe.

Drzewo rozbioru zapytania jest przeglądane w głąb. Algorytm przeszukiwania jest rekurencyjny. Dla pierwszego wywołania należy podać korzeń drzewa:

```
HandleComplexQuery(node) //dla pierwszego wywołania node jest korzeniem
if node is of class Literal then
  return Adjiman(node)
end if
if node is of class ComplexQuery then
  if node.operator is ALTERNATIVE then
    for n in node.children do
      answer = Adjiman(n)
      if answer is true then
        return true
      end if
    end for
    return false
  end if
  if node.operator is CONJUNCTION then
    for n in node.children do
      answer = Adjiman(n)
      if answer is false then
        return false
      end if
    end for
    return true
  end if
end if
```